Parallelizing Dijkstra's Algorithm With Java Threads

Tyler Knapp and Shane Bourdeau

9 December 2024

Abstract

Dijkstra's algorithm is one of the most fundamental algorithms in graphing and mapping. Given a weighted and connected graph, Dijkstra's algorithm finds the shortest path between two nodes. With its efficiency, the algorithm is used for large-scale graphs and has many real-world applications such as IP routing and building road networks. Our study will focus on parallelizing the algorithm through the priority queue using Java Threads and analyzing its performance. By introducing threading to Dijkstra's Algorithm, we look to improve speedup and reduce runtime, although the shortest path found may not be the most optimal. We also looked at different ways to parallelize the algorithm and key challenges to it, such as thread synchronization and overhead. Through testing on various graph sizes, we observed that parallelizing the priority queue can slightly reduce runtime for large graphs but excessive thread counts can lead to overhead. We hope our findings could contribute to more efficient ways to implement Dijkstra's algorithm.

Introduction

Dijkstra's algorithm is fundamental to mapping because it finds the shortest path between two nodes. The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found, and then adds it to the priority queue. This process continues until the destination vertex is reached, or all reachable vertices have been visited (What Is Dijkstra’s Algorithm?).

The priority queue is one of the most important parts of the program. In the serial algorithm you insert the starting vertex into the priority queue with a distance of zero, then extract the node with the minimum distance from the priority queue, update the distances of its neighbors if a shorter path is found, and then insert the updated neighbors into the priority queue (What Is Dijkstra’s Algorithm?).

We broke our project down into parts such as how we explored coding the program, effectiveness, timings and what we learned. Our paper will mainly examine the implementation using Java threads and follow the following outline:

1) Introduction explores how we coded the program
2) Effectiveness with our software
3) What we learned after coding the algorithm
4) Timings from the study
5) Parallelizing with other languages
6) Conclusion

Section 1

Using a serial version of Dijkstra's Algorithm as a baseline, we made modifications to parallelize the program using Java threads. One way we considered parallelizing the program was having each thread compute its own instance of the program independently using random start and ending points. However, we decided not to use this method because it introduced challenges. For example, if one thread chooses two points that do not connect, the program would stall leading to inefficiency.

While we could have used a spanning tree rather than the shortest points, we decided not to take this approach as it had several disadvantages. For example, Threads would need to share updates on the shortest path. This constant synchronization would lead to overhead when accessing the priority queue. Our goal is to have an efficient algorithm that improves speedup and runtimes, and this would cause us to have a limited speedup on the algorithm.

Another method we considered was just parallelizing the priority queue, a key component of the algorithm. By parallelizing the priority queue, each thread would get its own access to the queue and add items to compute the shortest path. We chose this method because we felt it's the most effective way to parallelize the program as you can update the distance to a vertex's neighbors concurrently. While we found this approach to be more efficient, there were disadvantages such as the fact that it does not compute the absolute shortest path, but rather gives an approximation of the shortest path.

The worker thread class in our program is responsible for parallelizing the priority queue. Each thread repeatedly calls poll() to process the next edge from the queue. When an edge is processed, the thread checks if the start vertex has been visited, if it has then it is skipped. If it

has not been visited, it is added to the result map which stores the shortest path from the start vertex to each destination. The threads are synchronized to make sure only one thread updates the map at a time. The threads then add all of the neighbors of the starting vertex to the priority queue if they haven't been visited already, this is also synchronized. Then the threads repeat the process until it finds the end vertex or all of the vertices have been visited. The code snippets below show this algorithm. Shared access to the priority queue allows the threads to explore different parts of the graph concurrently, making Dijkstra's algorithm parallelized.

```
public void run() {
    while (!Dijkstra.found.get()) {
                    //if there are no more edges to process, continue
        PQEntry nextPQ = Dijkstra.pq.poll();
        if (nextPQ == null) {
            continue;
        }
                    //if the edge is null, continue
        HighwayEdge nextEdge = nextPQ.lastEdge;
        if (nextEdge == null) {
            continue;
        }
                    //if the destination vertex is null, continue
        HighwayVertex destVertex = Dijkstra.g.vertices[nextEdge.dest];
        if (destVertex == null) {
            continue;
        }
                    //if the destination vertex is already visited, continue and synchronize
                    synchronized (destVertex) {
            if (destVertex.visited) {
                continue;
            }
            destVertex.visited = true;
        }
```

```
            //add the edge to the result map and lock map to prevent thread racing
 Dijkstra.resultLock.lock();
 try {
     Dijkstra.result.put(destVertex.label, nextEdge);
 } finally {
                    //unlock the map
     Dijkstra.resultLock.unlock();
 }
            //if the destination vertex is the destination, set found to true and return
 if (destVertex.vNum == Dijkstra.dest.vNum) {
     Dijkstra.found.set(true);
     return;
 }
            //add the edges of the destination vertex to the priority queue
 HighwayEdge e = destVertex.head;
 while (e != null) {
     if (Dijkstra.g.vertices[e.dest] != null && !Dijkstra.g.vertices[e.dest].visited) {
                        //syncrhonize the priority queue to prevent thread racing, and print out what each thread added to the queue
         synchronized (Dijkstra.pq) {
             Dijkstra.pq.add(new PQEntry(nextPQ.totalDist + e.length, e));
                                System.out.println("Thread " + threadId +" to " + Dijkstra.g.vertices[e.dest].label + " via " + e.label);
         }
     }
                //move to the next edge
     e = e.next;
 }
```

Section 2

Initially, we were going to code our program in C. After some time working on the serial version in C, we decided to pivot to Java threads as we are more comfortable in Java and using shared memory instead of message passing. For example, java has synchronized blocks and locking/unlocking that makes it easier to share the priority queue. Another reason is we both have different operating systems, which makes it more challenging to program with Pthreads, as Pthreads are platform dependent because you need a UNIX like system. Java threads are platform independent which makes it easier for us to work. Multithreading gives us more control in parallelization compared to message passing.

There are several features for Java threads that are not in other languages such as C. For example, Java uses garbage collection so pointers are not used which makes the program easier to code. This eliminates the need for any pointers like in C. With the elimination of pointers, this reduces chances of memory management errors. Another advantage are the mutexes and barriers for synchronization. These tools are necessary as they coordinate thread operations and prevent race conditions which could give us incorrect approximations. We see these tools as a way to enhance our program stability.

Another advantage Java has over C, is Java has a big library for multithreading which has thread safe data structures such as the ConcurrentLinkedQueue which was used in our program. The ConcurrentLinkedQueue is a thread safe queue with built-in methods for queues such as peek(), poll(), and pop(). This makes programming the priority queue faster because we don't have to make those methods from scratch. It is used when many threads have access to a common collection (oracle). Another useful thing Java has is the ArrayList class. It lets us dynamically add and remove elements from the queue. As with the ConcurrentLinkedQueue, the built in functions of this class make it easier to code because the methods are already written. These considerations made us choose Java threads over parallel software in C.

Section 3

While parallelizing the priority queue, we deepened our understanding of how to manage java threads. One of the issues we encountered during the process was thread racing. The threads would keep adding items to the shared priority queue and this would lead to null pointer exceptions. Thread racing highlights the importance of thread synchronization. To resolve this issue, we implemented mutex locking and unlocking. These ensured that only one thread could

access the priority queue at a time. The program would lock the graph before a thread puts the item in the queue and unlock it after. This is shown in the code snippet below:

```
//add the edge to the result map and lock map to prevent thread racing
Dijkstra.resultLock.lock();
try {
    Dijkstra.result.put(destVertex.label, nextEdge);
} finally {
    //unlock the map
    Dijkstra.resultLock.unlock();
}
```

Using this strategy prevented any concurrency to the priority queue, eliminated null pointer exceptions, and ensured data consistency. The locking and unlocking is similar to pthreads synchronization. By locking the priority queue during each threads operation, only one thread can add to the priority queue at one time as well as making sure no other thread can add to it. This not only helped us resolve the null pointer exceptions, but deepened our understanding of thread safety and synchronization.

Section 4

For this study, we collected timings on different size maps of various sizes on Noreaster. Noreaster is a server equipped with 20 physical cores and 20 virtual cores, making it a good choice for testing our program as its infrastructure allows for efficient computing of parallel tasks. We were curious to see our times scale with changes in thread count and various vertices and edge sizes. We choose to use between 1,2,4,8,16,32,64 and 128 threads for each map to see if there is any overhead.

Graph Details				
Name	**|V|**	**|E|**	**Start**	**End**
RPI	73	85	NY2@CR139	NY378/US9
London	2269	3425	A414/A1019/A1169	B2036/B2037
Wolf Creek	128	128	CO149@SanJuanDr	CO17Ant@CRD.5

Flagstaff	349	413	US160@BIA16	AZ89@ESCWAY
NY	6774	8077	NY18@OldLakeRd	NY27@End
United States	186250	221643	US82Bus_E/US271Bus_S	MA10@EarSt

Thread Timings

	RPI		London		Wolf Creek		Flagstaff		NY		USA	
Threads	**MS**	**Mi**	**MS**	**Mi**	**MS**	**Mi**	**MS**	**Mi**	**MS**	**Mi**	**MS**	**Mi**
1	100	9.59	305	68.92	111	184.64	181	282.82	229	582.23	4826	1853.50
2	94	9.59	311	64.92	116	184.64	162	282.82	212	582.23	4808	1857.72
4	92	9.59	312	64.92	1114	184.64	181	233.84	193	593.86	5002	1871.49
8	93	9.59	286	64.92	157	184.64	187	239.22	204	580.36	6859	1846.38
16	98	11.57	303	64.92	559	184.64	192	238.82	205	595.92	6006	1859.99
32	99	9.59	307	61.14	513	184.64	172	272.29	354	591.99	5891	1910.97
64	108	11.27	302	88.76	1837	184.64	198	272.29	238	636.88	14213	1880.99
128	107	6.61	318	64.92	486	184.64	232	272.29	290	587.94	65829	1973.67

There are several limitations to the program. The first being that the serial program is very fast using TMG graphs from METAL, so there won't be much of a meaningful speedup. We noticed this when running the United States map end to end. It has over 18,000 vertices and it took about 5 seconds to run the serial version. It is possible that it can take longer with a bigger map but that increases the risk of there being a disconnection between nodes and Dijkstra's algorithm does not work with disconnected graphs.

The other limitation is there is not a big speedup with multiple threads. This is because the threads don't have a lot of work to do before locking and unlocking which causes overhead. A lower number of threads is better, specifically two, four, or eight threads, because we get too much overhead with a large number of threads. This is shown in the NY graph. The shortest path is also approximated when using more than one thread. The best number of threads, whether it is two, four, or eight, depends on the graph. We get a different answer in the parallel version with each run, as the shortest path found is close to the serial version, which is always the correct answer. With parallelizing the program, the sacrifice we make for speedup is an approximation of the shortest path. However, sometimes the parallel version gives us the correct shortest path. Having four threads for this map is the most efficient number to solve the problem, although it can be slower than the one thread version.

Section 5

Our way is not the only way to parallelize Dijkstra's algorithm. There have been other attempts to parallelize the algorithm with other parallel computing software. One example is from a St. Cloud State University paper where Dijkstra's algorithm was parallelized using MPI and OpenMP. In the paper, the serial algorithm used an adjacency matrix instead of a priority queue. The adjacency matrix represents a graph. In the OpenMP implementation, the threads have their own local distances to the vertices. The threads share visited vertices, the number of threads, the minimum and maximum distances to each vertex. The graph is divided into N threads. Each thread searches for the unvisited vertex with the smallest distance in that thread's range of vertices. After the local distances and vertices are found the threads compute the global minimum distance and closest vertex one thread at a time in the critical section. After the threads are synchronized, it marks the vertex as visited and updates the distance to its neighbors. It repeats the process until all of the vertices are visited (He, 56).

The MPI implementation is more complicated than the OpenMP implementation. The graph is divided into N processes and each process does its own local portion of the graph. The adjacency matrix is distributed using `MPI_Scatter`. The function that parallelizes Dijkstra's algorithm is called SingleSource. It initializes the local distances from the starting vertex, then it marks which vertices have been visited. In the main loop of Dijkstra's algorithm, every local vertex is iterated over and each vertex finds its own local minimum distance from the starting vertex. After each local minimum distance is found, the program uses MPI_Allreduce to

compute the global minimum. After the global minimum is computed, each process updates its own tentative distances for their vertices. This repeats for N times where N is the number of vertices. After all the vertices have been visited, the results are gathered to the root process with MPI_Gather (He, 48).

This paper found similar results to us. The best number of threads/processes to use was 2,4, and 8, with the most speedup in the experiment being with 4 threads. A larger number of threads caused too much overhead and the speedups were non linear. (He, 38).

Section 6

In this study, we explored how to parallelize Dijkstra's Algorithm with focusing on the priority queue in Java Threads. While we had some disadvantages such as a less optimal path and overhead, we feel we have an efficient parallelized version of the algorithm that demonstrates speed and runtime improvements. By using some of Java threads multithreading features such as thread locking and unlocking, we were able to prevent thread racing and null pointer exceptions in the priority queue. Our results show that the optimal thread count is dependent on the size and complexity of the graph, with two, four or eight threads having the best performance. This project enhanced our understanding of multithreading and synchronization and algorithm optimization. While challenges such as thread synchronization overhead remain, this study serves as a stepping stone toward more efficient implementations of Dijkstra's algorithms in the world of parallel computing.

Works Cited

GeeksForGeeks. “What Is Dijkstra’s Algorithm?: Introduction to Dijkstra’s Shortest Path Algorithm.” *GeeksforGeeks*, GeeksforGeeks, 9 May 2024, www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/#.

He, Mengqing. *Parallelizing Dijkstra’s Algorithm*, Dec. 2020, repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1044&context=csit_etds.

“The Metal Project.” *Map-Based Educational Tools for Algorithm Learning (Metal)*, courses.teresco.org/metal/. Accessed 7 Dec. 2024.

Oracle. “ConcurrentLinkedQueue” *Oracle (Java Platform SE 8), 30 Sept. 2024,* docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html.